DIMACS Series in Discrete Mathematics
and Theoretical Computer Science
Volume **27**, 1996

Complexity of Restricted and Unrestricted Models of Molecular Computation

Erik Winfree[1]
Computation and Neural Systems
California Institute of Technology
Pasadena, California 91125, USA
winfree@hope.caltech.edu

Abstract

In [9] and [2] a formal model for molecular computing was proposed, which makes focused use of affinity purification. The use of PCR was suggested to expand the range of feasible computations, resulting in a second model. In this note, we give a precise characterization of these two models in terms of recognized computational complexity classes, namely branching programs (BP) and nondeterministic branching programs (NBP) respectively. This allows us to give upper and lower bounds on the complexity of desired computations. Examples are given of computable and uncomputable problems, given limited time.

1 Introduction

Molecular computation, as introduced by [1], provides a new approach to solving combinatorial inverse problems, where we are interested in computing $f^{-1}(1)$ for n-bit strings $\underline{x}$ and boolean function f. Instances of NP-complete problems can be expressed in this form; for example 3-SAT. Adleman's technique involves using individual DNA strands to represent potential solution bit-strings $\underline{x}$, then operating on a test tube containing all possible solutions to separate those which satisfy f from those which don't. In many instances, the number of sorting operations required is a low-order polynomial in n, suggesting that

[1]This work is supported in part by National Institute for Mental Health (NIMH) Training Grant # 5 T32 MH 19138-05; also by General Motors' Technology Research Partnerships program.

– given exponential space to store the DNA[2] – hard combinatorial problems can be solved efficiently with this technique.

It was not immediately clear, however, what class of boolean functions f could be efficiently inverted. In a clarifying paper by Lipton [9], it was shown that if f can be represented as a size L formula of AND-OR-NOT (AON) operations, then f can be inverted using $2L$ molecular steps using affinity purification[3] only. Lipton suggested further that the use of PCR[4] to duplicate the contents of a test tube would allow an even greater class of functions to be inverted using molecular computation. In this note we follow his program and characterize exactly to what extent PCR helps, in terms of known complexity classes.

As individual steps can take on the order of 15 minutes to an hour, small differences in complexity quickly make the difference between feasible and infeasible experiments. Thus it is of importance to characterize the complexity of these models of molecular computation as carefully as possible. Classes such as "polynomial-size" are too rough to be really useful – we really want to know exactly what polynomial it is.

After defining the two models of molecular computation, we will demonstrate their correspondence with branching programs, and conclude with a few implications of the correspondence.

2 Abstract Models of Molecular Computation

We use the models described in [9] and [2], and use similar notation. These models assume perfect performance of each operation, although in practice the molecular biology techniques are known to be somewhat unreliable. Initial comments on this aspect of the models, and other practical matters, can be found in [2], and will not be address here.

The Restricted Model:

[2]In this note we grant that $O(2^n)$ volume is "reasonable". Using substantially more DNA, e.g. to search over additional non-deterministic variables, is considered "cheating". In other words, the question being addressed here is, "Given a fixed amount of DNA, what functions can we easily solve?"

[3]Or some equivalent technique.

[4]Or some equivalent technique.

A *test tube* is a set of molecules of DNA encoding bit-strings of length n. We operate on test tubes as follows:

- *Separate.* Given a tube T and an index[5] i, produce two tubes $+(T,i)$ and $-(T,i)$, where $+(T,i)$ contains all strings where bit i is set, and $-(T,S)$ contains all strings where bit i is cleared. Tube T is destroyed.

- *Merge.* Given tubes T_a and T_b, pour T_b into T_a thereby making $T_a \leftarrow T_a \cup T_b$. Tube T_b is destroyed.

At the end of the computation[6], when we presumably have a single test tube containing all strings in $f^{-1}(1)$, we can use the following operation to sequence the strings $\underline{x}$ in the test tube, as described in [2]:

- *Detect.* Given a tube T, say 'yes' if T contains at least one DNA molecule, and say 'no' if it contains none. Tube T is preserved.

A *program*[7] is a sequence of operations on labelled test tubes. Each statement is of the form:

$$\langle +(T_a,i) \rightarrow T_b; -(T_a,i) \rightarrow T_c; \rangle,$$

where the arrow means "is to be merged with". In other words, one separation and two merges occur for every statement (but note that T_b or T_c may be empty prior to the merge). For clarity, programs can be shown diagrammatically (see Figure 1). At the beginning, all test tubes are empty except for T_1, which contains all 2^n DNA strands encoding all possible input vectors $\underline{x}$. If at the end of the program execution there is a test tube containing exactly those bit strings which satisfy f, then we say say the program has inverted f, or has solved f. The *size* of a program is considered to be the number of statements (here

[5]We consider only the case where one variable at a time is tested. More sophisticated operations where multiple DNF minterms are tested simultaneously (see [6]) require more lengthy preparation; thus we argue that the single variable case is not unreasonable for measuring complexity.

[6]We do not consider here whether *Detect* could be used to advantage in the middle of a computation.

[7]The class of programs as given here is slightly different from that given in [2]. In particular, we insist that a labelled test tube is not re-used after its contents have been used (i.e. "destroyed"). The differences are merely a matter of notation, and inconsequential.

Separate operations) in the program. Since programs are considered to be executed sequentially, the size of a program to invert f is often refered to as the time to solve f. The *width* of a program is the maximum number of test tubes co-existing at any given time.

Figure 1:

Implementing an arbitrary symmetric function in $\frac{n(n+1)}{2}$ *separations (restricted model).*

$$f(\underline{x}) = \text{“}0 < \sum_i x_i < 4\text{”}$$

Given $T_1 = \{0,1\}^n$.

$\langle +(T_1, 1) \rightarrow T_3; -(T_1, 1) \rightarrow T_2; \rangle$

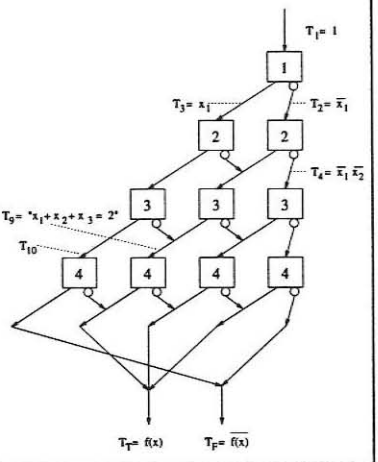
$\langle +(T_2, 2) \rightarrow T_5; -(T_2, 2) \rightarrow T_4; \rangle$

$\vdots$

$\langle +(T_9, 4) \rightarrow T_T; -(T_9, 4) \rightarrow T_T; \rangle$

$\langle +(T_{10}, 4) \rightarrow T_F; -(T_{10}, 4) \rightarrow T_T; \rangle$

Return T_T.



The Unrestricted Model:

The unrestricted model allows one addition type of operation during the computation:

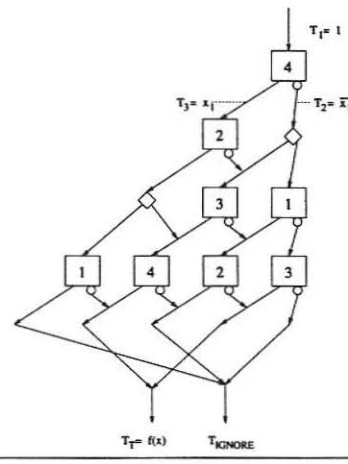
- *Amplify.* Given a tube T produce two tubes T_1 and T_2 with contents identical to T. T is destroyed.

Programs for the unrestricted model consist of statements similar to those for the restricted model, but with the additional form:

$$\langle T_a \rightarrow T_b, T_c; \rangle$$

Here the arrow means, “is to be copied into.” Unrestricted model programs can also be shown diagrammatically (see Figure 2).

Figure 2:
Implementing a random function using the unrestricted model. ($f(x) = x_4(x_2 + x_3) + \overline{x}_4(x_1\overline{x}_2 + \overline{x}_1x_3 + \overline{x}_3x_2)$).



We might expect that the unrestricted model is siginificantly more powerful than the restricted model. This expectation is quantified and explored in what follows.

3 Branching Programs

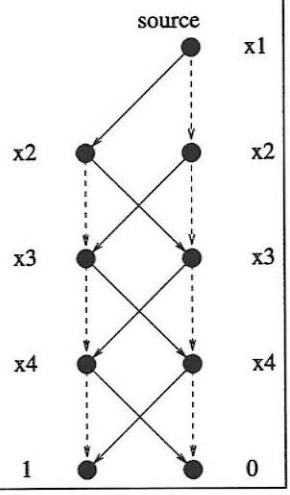
Since branching programs are not as familiar a model as formulas, finite-state automata, circuits, Turing machines, etc., it is worthwhile to present an exact definition here. We quote from [16], p. 414:

> A branching program (BP) is a directed acyclic graph consisting of one source (no predecessor), inner nodes of fan-out 2 labelled by Boolean variables and sinks of fan-out 0 labelled by Boolean constants. The computation starts at the source which is also an inner node. If one reaches an inner node labelled by x_i, one proceeds to the left successor, if the i-th input bit a_i equals 0, and one proceeds to the right successor, if a_i equals 1. The BP computes $f \in B_n$[8] if one reaches for the input a a sink labelled by $f(a)$.

[8] B_n is the set of all n-input boolean functions.

The size of a BP is the number of inner nodes. Many measures of BP have been studied, especially depth and width.

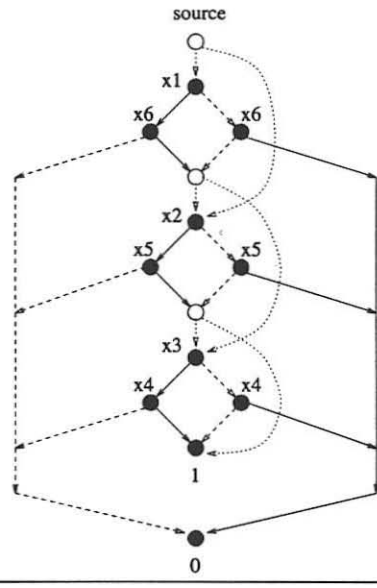
Figure 3:
Implementing PARITY of 4 variables using a branching program of width 2.



We follow [13] in defining a nondeterministic branching program (NBP): we additionally include unlabelled "guessing nodes" of fan-out 2 where both branches are allowed[9]. The NBP computes $f \in B_n$ if by some allowable path one reaches a sink labelled 1 for all $a \in f^{-1}(1)$. The size of an NBP includes the guessing nodes. BP and NBP may be viewed pictorially, as in Figures 3 and 4, in which the designations "left" and "right" are replaced by "dotted-line" and "solid-line" respectively.

[9]This definition of NBP coincides exactly with Meinel's 1-time-only nondeterministic branching programs. His more general definitions seem not to be useful in the context of molecular computing.

Figure 4:
Implementing a function using a nondeterministic branching program. $f(x)$ = *"$\underline{x}$ is palindromic except for isolated (non-adjacent) errors".* $NBP(f) \leq 2n + 2$.



4 Correspondence of Models

Restricted Model $\sim$ *Branching Programs*

In this section we show that the class of functions which the restricted model can invert in a given time are exactly those functions computed by a branching program of the same size.

Examining Figures 1 and 3, it is clear that not much needs to be proved. The models are essentially identical, except for interpretation. Each separation step corresponds to an inner node of the BP. A strand of DNA corresponds to an input vector for the BP. In summary:

1. If restricted model program P solves f in k steps, then there is a BP G which computes f and is of size k.

2. If BP G computes f and is of size k, then there is a restricted model program P which solves f in k steps.

A single strand of DNA will flow through the test tubes of a restricted model program exactly in the order of inner nodes executed

by the associated BP running on an equivalent input vector[10]. Since all possible strands are run in parallel, those that end up in the ouput test tube T_T are exactly the inputs that the BP accepts; i.e. $f^{-1}(1)$.

Unrestricted Model ~ Nondeterministic Branching Programs

In this section we show that the class of functions which the unrestricted model can invert in a given time are exactly those functions computed by a nondeterministic branching program of the same size.

Examining Figures 2 and 4, it is clear that not much needs to be proved. We additionally associate *amplify* statements with guessing nodes in the NBP. Just to be clear, we show:

1. If unrestricted model program P solves f in k steps, then there is a NBP G which computes f and is of size k.

2. If NBP G computes f and is of size k, then there is a unrestricted model program P which solves f in k steps.

We use essentially the same argument as above. However now we say that the set of test tubes which a DNA strand passes through is the same as the set of nodes of the NBP which *could* be activated by the associated input vector. Thus the output test tube contains all strands which *could* cause the NBP to accept; i.e. $f^{-1}(1)$.

5 Corollaries and Conclusions

We now have a theoretical handle on precisely what can and cannot be computed by the restricted and unrestricted models. First, by looking at the polynomial size complexity hierarchy, we can separate the classes of functions solvable by the DNA models.

Many useful results follow immediately from the literature on branching programs. Here is a brief sampler:

- poly-size BP are equivalent to log-space non-uniform TM[11] [11].

[10]The author is reminded of some friends who needed to transfer a lot of graphics images from San Francisco to Los Angeles. They considered using ftp over the internet, but on second thought realized it would be faster to put the data in their car and drive, so they did. We are doing the same thing here: We physically move a bunch of DNA through the virtual CPU, one gate at a time – but lots of data simultaneously.

[11](N)TM = (nondeterministic) Turing machine.

- poly-size NBP are equivalent to log-space non-uniform NTM [11].
- poly-size circuits[12] are equivalent to poly-time non-uniform TM [16].
- thus poly-size BP $\subseteq$ poly-size NBP $\subseteq$ poly-size circuits, where the inclusions are believed to be proper.
- poly-size, constant-width BP are equivalent to log-depth circuits [3] [10].
- $\sqrt[3]{C(f)} \preceq NBP(f) \preceq BP(f) \preceq L(f)$ [13][13].
- $\frac{C(f)}{3} \leq BP(f) \leq L(f) + 1$ [16][14].

With each of these results there is typically an efficient simulation [12]. Other known linear simulations by branching programs include finite-state automata (FSA) and 2-way finite-state automata [3].

As mentioned earlier, results on polynomial equivalence are only of theoretical and not practical relevance. We would like more exact bounds on the complexity of implementing specific functions. The literature on branching programs gives us some such bounds, although admitedly the knowledge is very incomplete. Some known bounds[15] for a few functions[16] are summarized in Table 1.

[12]In this note we consider circuits where gates are fan-in 2, arbitrary fan-out, and have arbitrary logic.

[13]C(f) is circuit size, L(f) is AON formula size, etc. $F \preceq G$ means $F = O(G)$.

[14]Note this construction for formulas is better than that given in [9].

[15]See especially [16]: pp. 76, 85, 143, 243, 247, 261, 440; [13]: pp. 50, 51; [8]: pp. 793-797. Note Razborov incorrectly quotes the BP lower bound on MAJORITY [4]. The upper bound comes from [14]. The upper bound on formulas for symmetric functions follows directly from the upper bound Wegener gives for MAJORITY. The upper bound on circuits for DISTINCT comes from a simple application of SORT, followed by adjacent comparisions; a better bound may be achievable. The upper bound on NBP for symmetric functions uses a construction by Lupanov for switching-and-rectifier circuits (see [13]); the construction also works for NBP.

[16]Let $m = \frac{n}{2logn}$, $|\underline{X}_i| = 2logn$ and DISTINCT$(\underline{X}_1, \ldots, \underline{X}_m) = 0$ iff $\exists i \neq j$ s.t. $\underline{X}_i = \underline{X}_j$. MAJORITY$(\underline{x}) = 1$ iff $|x| \geq \frac{n}{2}$. PARITY$(\underline{x}) = 1$ iff $|x| \equiv 1 \bmod 2$. f is SYMMETRIC if f depends only on $|\underline{x}|$, the number of 1's in $\underline{x}$. The lower bounds are for almost all symmetric f.

Table 1. *Lower and upper bounds on complexities under known models for various functions.*

function f_n	$L(f)$	(AON)	$BP(f)$	
PARITY	n^2	n^2	$2n-1$	$2n-1$
DISTINCT	$\Omega(\frac{n^2}{\log n})$	$O(n^2 \log n)$	$\Omega(\frac{n^2}{\log^2 n})$	
MAJORITY	$\Omega(n^2)$	$O(n^{3.37})$	$\Omega(\frac{n \log n}{\log \log n})$	$O(n \log^3 n)$
SYMMETRIC	$\Omega(n \log \log n)$	$O(n^{4.37})$	$\Omega(\frac{n \log n}{\log \log n})$	$O(\frac{n^2}{\log n})$

function f_n	$NBP(f)$		$C(f)$	(B_2)
PARITY		$2n-1$	$n-1$	$n-1$
DISTINCT	$\Omega(\frac{n^{3/2}}{\log n})$		$\Omega(n)$	$O(n \log n)$
MAJORITY	$\Omega(n \log \log \log^* n)$		$\Omega(n)$	$O(n)$
SYMMETRIC		$O(n^{3/2})$	$\Omega(n)$	$O(n)$

6 Discussion

Do we gain anything by using the *amplify* operation? Theoretically, yes, but very little. Contrary to the suggestion in [9][17], we probably cannot invert functions defined by circuits in linear size. Furthermore, in addition to concerns about the reliability of PCR, we should realize that each *amplify* at least doubles the volume of DNA that we have to handle. After just a few such operations, we could practically be unable to continue the computation. For example, if we conclude for practical reasons that 2^{50} molecules of DNA are the most we can handle in one test tube, then we must be very careful not to exceed this limit when merging the products of amplification[18].

The restricted and unrestricted models of molecular computation are still a long way from allowing us to invert algorithmically defined boolean functions. It seems that new molecular operations are necessary if we need this functionality – for example, operations which modify DNA during the computation, such as Adleman's memory

[17] It appears that Lipton realized this shortly after distributing his draft. He later characterizes his constructions in terms of *contact networks*, which are related to branching programs (personal communication).

[18] On a similar note, even the restricted model can solve f computed by Meinel's more general NBP model, simply by using 2^m times more DNA volume when there are m non-deterministic variables. This allows computation as efficient as circuits, but at the cost of ridiculous amounts of DNA.

model [2] which can be implemented via site-directed mutagenesis, Beaver's Turing Machine simulation [5] which uses similar mechanisms, or Boneh's *Append* [7], perhaps the simplest and most elegant extension.

Acknowledgments

The author would like to thank Paul W. K. Rothemund, Sam Roweis, and Matthew Cook for their stimulating discussion. Thanks especially to Jehoshua Bruck for pointing me to previous literature on branching programs. Thanks to my advisor John Hopfield for his support and encouragement.

Bibliography

[1] Adleman, Leonard, Molecular computation of solutions to combinatorial problems, *Science* 266:1021–1024 (Nov. 11) 1994.

[2] Adleman, Leonard, On Constructing a Molecular Computer, draft, Jan. 11, 1995.
(ftp://usc.edu/pub/csinfo/papers/adleman/molecular_computer.ps)

[3] Barrington, David A., Bounded Width Branching Programs, PhD Thesis, 1986, Massachusetts Institute of Technology, TR# MIT/LCS/TR-361.

[4] Babai, L., P. Pudlák, V. Rödl, E. Szemeredi, Lower Bounds to the Complexity of Symmetric Boolean Functions, *Theoretical Computer Science* 74 (1990) 313-323.

[5] Beaver, Don, A Universal Molecular Computer, Draft Feb. 6, 1995.
(http://www.cse.psu.edu/~ beaver/research/molec.html)

[6] Boneh, Dan, Christopher Dunworth, and Richard J. Lipton, Breaking DES Using a Molecular Computer, sumbitted to IEEE COMPUTER.
(http:/www.cs.princeton.edu/~ dabo/papers/biocomp.ps)

[7] Boneh, Dan, C. Dunworth, R. Lipton, and J. Sgall, On Computational Power of DNA, to appear.

[8] Boppana, R. B. and M. Sipser, The Complexity of Finite Functions, in *Handbook of Theoretical Computer Science*, ed. J. van Leeuwen, pp. 757-804, 1990, Elsevier Science Publishers B. V.

[9] Lipton, Richard, Speeding up computations via molecular biology, draft Dec. 9, 1994.
(http://www.cs.princeton.edu/~ rjl/bio.ps)

[10] Lipton, Richard, Subquadratic Simulations of Circuits by Branching Programs, in 30^{th} *Annual Symposium on Foundations of Computer Science*, pp. 568–573, 1989, IEEE Computer Society Press.

[11] Meinel, Christoph, *Modified Branching Programs and Their Computational Power, LNCS 370.* 1989, Springer-Verlag.

[12] Pudlák, Pavel, The Hierarchy of Boolean Circuits, *Computers and Artificial Intelligence*, 6 (1987), No. 5, pp. 449–468.

[13] Razborov, Alexander A., Lower Bounds for Deterministic and Nondeterministic Branching Programs, in *Fundamentals of Computation Theory, LNCS 529*, pp. 47–60, 1991, Springer-Verlag.

[14] Sinha, Rakesh Kumar, and Jayram S. Thathachar, Efficient Oblivious Branching Programs for Threshold Functions, in *Proceedings of the 35th Symposium on Foundations of Computer Science*, pp. 309–317, 1994.

[15] Wagner, K. and G. Wechsung, *Computational Complexity*, D. Reidel Publishing Company, 1986.

[16] Wegener, Ingo, *The Complexity of Boolean Functions*, John Wiley & Sons, 1987.